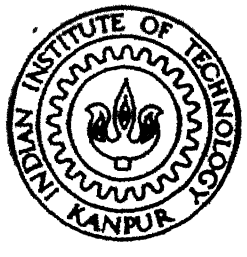
PROCESS MIGRATION FOR LOAD BALANCING

by

M. L. Prasanna Kumar Reddy

CSE
1996
M
RED
DRO



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

August, 1996

Process Migration for Load Balancing

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

M. L. Prasanna Kumar Reddy

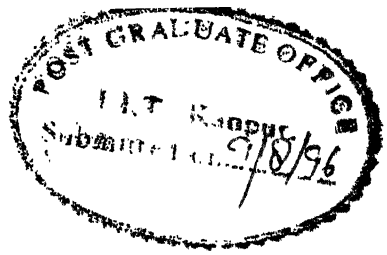
to the

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

August, 1996.



CERTIFICATE

This is to certify that the work contained in the thesis entitled "*Process Migration for Load balancing*" by "*M. L. Prasanna Kumar Reddy*" (Roll No: 9411125), has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Rajat Moona
Associate Professor
Department of Computer
Science & Engineering,
Indian Institute of Technology,
Kanpur.
August, 1996

CENTR RY
Acc. No. A 123357

C.S.E-1996-M-RED-PRD

Acknowledgments

I profoundly thank my thesis supervisor **Dr. Rajat Moona** for his constant guidance throughout this work. I thank Dr. Barua for teaching Distributed Systems and suggesting this problem. Dr. S. Biswas is the sole inspiration for my academic career in IITK. I thank Dr. Deepak Gupta for his timely advice and suggestions regarding the thesis.

My special thanks go to Ramakrishna(_a_a), Siva, Anil and Sricharan for their friendship and encouragement. Srivatsa is our Krishna Murthy(Jiddu). Chaveli and nyal are my phdian friends. I thank my classmates dileep, nandan, jayaram, suresh, raghuram, chap, sudhakar, anand and others. I can't forget my friendly days with my juniors gvrk, chitti, mrl, panku, pvn, kommu, dsri, gsri, praveen, and kala. I thank the CSE Lab staff, especially Brahmaji for his help in working with the systems . I thank Sri Ramatur for his regular service in dept library.

Finally, I thank my parents and brother for their love and encouragement.

Abstract

Process migration is relocating a process from one machine to another machine during its execution. It is a mechanism to utilize the idle computing power that is being wasted in a distributed system. In this thesis, distributed process subsystem, that provides migration facilities, is developed on SunOS. File maintenance, maintenance of process relationships and a load balancing policy are developed for migration. The memory management used for this system is from *rfork* model, already implemented.

Contents

List of Figures

Chapter 1

Introduction

Migration is a potential concept of distributed systems through which valuable computational power can be utilized. Migration of a process is the transportation of a running process from one machine to another. The most common objective of using process migration is homogeneous distribution of load on all machines in a network.

1.1 Objective

Our primary goal is to provide migration facilities useful for load balancing. Apart from this main objective, there are certain guide lines which have been followed, as far as possible, in developing the model.

- **Transparency:** After migration, process should behave in the similar manner as before migration. Additionally, process should see the outside world same as before and outside world should see the process as before.
- **Compatibility:** The approach should be compatible with Unix semantics and should be achieved with minimal modifications in Unix kernel. Until the model gets stabilized, it should support conventional Unix semantics.

Our approach should give maximal support to processes, whose computation is high and interaction with outside world is low. At the same time, it should provide

environment to processes whose interaction with outside world is complex, as far as possible.

1.2 Method followed

We provide migration facilities by means of few system calls and migration server mechanism on top of *SunOS* kernel. Load balancing is provided by user level software and load daemon to calculate dynamic load on the system.

1.3 Motivation

Typical workstations show a low average CPU utilization and many idle intervals. Especially at late nights, a significant proportion of workstations on network is unused or under light load. These workstations have spare processing capacity, and can be used for users, who logged at other workstations and feel insufficient computing capability at their sites. By using the unused computing power at other sites, high system utilization can be achieved.

1.4 Organization of the report

The thesis has been organized as follows. In the next chapter, we present the related work. In chapter three, we explore the issues related to process migration. In chapter four, we provide the details about design. In chapter five, we present our implementation issues. Finally in the last chapter, we conclude the thesis and discuss some future work.

Chapter 2

Background

In this chapter, concepts of distributed systems and related work are presented.

Two advances in technology, namely the development of very powerful integrated circuits and the availability of high speed networks, led to the existence of computing systems composed of large number of CPUs.

2.1 Systems with multiple CPUs

These systems can be characterized as follows.

- **Multiprocessor system** consists of *tightly coupled hardware.*
- **Distributed system** consists of *loosely coupled hardware*, but *tightly coupled software.*
- **Network system** consists of *loosely coupled hardware* and *loosely coupled software.*

In loosely coupled hardware, there are autonomous computers that have no shared memory and communicate through messages using network. These are also known as NORMA (No Remote Memory Access) systems. In tightly coupled hardware systems, many CPUs share the same physical memory through their address spaces. Data transfer rate is low in loosely coupled hardware. In tightly coupled

software system, software provides the view of a single virtual machine, even though hardware is loosely coupled.

The only difference between distributed systems and network systems is in accessibility of resources using software. In network systems, users access resources explicitly. Distributed systems develop illusion to the users that entire network is a *virtual uniprocessor*. In distributed system, software conceals underlying hardware.

2.2 Definition of distributed System

A distributed system is a collection of independent computers that appear to the users of the system as a single computer system [Tanen95].

This definition implies two aspects of the system.

- **Hardware component:** Autonomous computers linked by a network.
- **Software component:** Distributed system software equipped at every site, that provides interface to other sites. This software is responsible for providing the look of a single computer to the user.

The term "distributed system" is applied for any software system working on a network of computers, in which resources are not localized to any particular computer and yet treated uniformly at all computers. In other words, underlying hardware is transparent.

2.3 Advantages of distributed systems

There are many advantages of distributed systems over conventional centralized systems.

- **Resource sharing:** When a facility is not available at a machine, a distributed system facilitates the use of the same facility, if it is available at some other site. For example, in a load balancing distributed system, idle computing resources are engaged with the work.

- **Transparency:** Users need not identify the machines where printers, files etc. are located. Users can access them irrespective of their position in a distributed system.

- **Fault tolerance:** Distributed systems can withstand crashes and failures of system components by maintaining redundancy and recovery.

- **Parallelism:** At the same time, many machines can be used for achieving single task and degree of parallelism can be achieved with economy.

- **Flexibility:** A distributed system may undergo changes and it should be flexible enough for importing them according to future demands. Micro kernels are designed with this motive. They provide minimal kernel support and leave many services to user level servers. Flexibility is of two types.

  Openness: Flexibility to replace or modify any component of system.

  Scalability: Flexibility to add new component to system.

  Here component is either software or hardware component.

2.4 Distributed computing

As the name implies, distributed computing is the utilization of the computing resources on many machines. Recently, it has become an attractive topic due to its capability to substitute the parallel computing. The fact is that buying computers of smaller size and networking them is cheaper than purchasing a massive parallel processor. The objectives behind distributed computing are parallelism and load balancing. PVM [Gei93] is example of a software interface to exploit parallelism across network of workstations and REM [Shoja87] is example of load balancing system.

Distributed computing can be either at granularity of tasks(inter-task computing) or at granularity of subtasks within a task(intra-task computing). Inter-task computation is more preferable for distributed systems due to low communication

and synchronization overhead. Intra-task computation is more suitable for multiprocessors. Developing distributed process subsystem leads to distributed computing of former type.

There are two approaches to distributed computing.

- User has to specify tasks or subtasks. Strictly speaking this approach is network computation. PVM [Gei93] is an example of such system.
- System selects subtasks or different tasks that can be distributed. For example Amoeba [MuSo90] is an operating system where tasks are distributed by the OS.

2.5 Impact of unix model on distributed OS

In past many distributed operating systems have been developed. Sprite [DoOu89],Mach [A Chorus [Coul94] and Amoeba [MuSo90] are few such examples. Many operating systems are not compatable with Unix operating system. Unix is a widely accepted system, but it has a drawback that it is not a distributed one. To understand its effect on other operating systems, consider the following.

While discussing Mach and its compatibility with Unix, Black et al [Bla95] says,

> no matter how novel its features, how elegant its design or how extensible its structure, it could succeed only if its Unix emulation was as good as or better than the native Unix on every platform on which it ran.

So a distributed software that is compatible with Unix is useful. One of the reasons behind Mach's stepping forward than others is it can emulate and provide environment of Unix.

Another approach is developing distributed extensions to Unix, instead of developing distributed operating systems. Interprocess communication facilities in a network, provided by BSD 4.3 [Leff89] make this approach feasible. File subsystem was the only subsystem that was implemented successfully in distributed manner in Unix OS. Other subsystems have not yet been implemented completely. Even in file

subsystem, complete transparency is not achieved. In this thesis also, an attempt is made to develop distributed layers on the top of Unix subsystems.

2.6 Components of distributed OS

Though one can not enumerate subsystems of a distributed system, as new components are always being added and it should be open and extensible, there are some major and minimal parts of OS that can be listed. Some major components are as follows.

- Process subsystem consists of process creation, scheduling and maintenance of processes.
- Interprocess communication, mutual exclusion.
- Memory subsystem consists of memory allocation, memory protection and management.
- File subsystem consists of file management and file access facilities.
- I/O subsystem, peripheral device handling.
- User authentication, access control and network security.
- Uniform naming scheme in network.

These components are however not completely independent of each other. Some examples of implementation of distributed subsystems are as follows.

- Amoeba's run server model [MuSo90]
- Andrew File system [Coul94]
- Sun NFS [Tanen95, RFC1094]
- Munin's distributed shared memory [Coul94]
- Iva's working page owner method [Coul94]

- Kerberos authentication system [SNS88, Stev92]
- Domain name service [Coul94]
- Mach's message-memory duality model [RYT87].

2.7 Work related to migration

Several migration facilities for the distributed systems are developed. Petri et al. [PeLi95] says,

> Due to the complex nature of the subject, all those facilities have limitations that make them usable for only limited cases of applications and environments.

Locus [WaPo83], Charlotte [ArFi89], V [Cher88], Sprite [DoOu89], MDX [Sch95] and MOSIX [Smit88] are operating systems that provide migration. Only Locus is the Unix compatible among these. Another credit of it is maintenance of pipe semantics using a network wide file system. Charlotte is first to divide migration into two layers. Sprite achieved high distributed functionality except for devices which are not known across machines. It has its own file system, but has a drawback that a migrated process depends heavily on original machine. An attempt to reduce memory transfer time is made in V Kernel. It implements pre-copying, technique similar to prepaging. MDX is complete object oriented implementation. MOSIX kernel contains one more interface layer, in addition to Charlotte layers.

Condor [LiMa92] is successful among the user level implementations. It wraps the `main()` routine and system calls in C program and maintains dummy processes. It uses `setjmp()` and `longjmp()` calls in Unix to get its memory image. Freedman [Freed91] is another user level system. It supports only memory transfer. Model given in [PeLi95] is similar to Condor, but it uses debugger interface provided by Unix.

Micro-kernel based migration is another attempt in research. Zayas [Zaya87] has done on Accent operating system, a predecessor of Mach OS [Acc86]. Mach based migration is another interesting work, since the OS is open and extensible. Lazy

copying is a technique in Mach that is helpful for migration. Actor based migration can be done in Chorus [Smit88].

Mach like operating systems are more suitable to add distributed subsystems like migration, since it is a message based OS. In a message based OS, processes interact by sending messages. By redirecting messages, new components of OS can be added or modified. In procedure oriented OSes like Unix and Sprite, interaction is by system calls, subroutines executing in kernel mode. Modifications to the OS require changes to these subroutines. Moreover Unix has monolithic kernel and it is therefore difficult to incorporate these changes.

Apart from the lack of complete Unix semantics, all the migration facilities suffer from one serious problem, that is lacking an efficient load distribution algorithm. This is an interesting area of research. The most cost-effective algorithm is, sending a random probe to a destination and taking decision [Tanen95].

2.8 Conclusion

Unix is designed for centralized systems and it satisfies all the requirements in such environments. However, it is difficult to modify it according to the emerging demands. It is not a distributed OS. Its structure is monolithic and it is not a message passing OS. Still all the research revolves around unix, since it has gained popularity of both users and researchers and is simple and sufficient enough for most of the current demands. It is also important to note that no other OS has satisfied the needs of such vast community upto now. So, we need facilitations like migration to be with Unix.

Chapter 3

Process Migration

Process migration is relocating the process during its execution. It is not the migration of code. It consists of maintenance and transfer of process state. Process state is composed of memory image of process, contents of registers and program counter, usage of resources, and communication to the outside world. For transferring a process from one machine to another, there are some phases.

3.1 Phases of migration

When a process migrates from machine **M1** to machine **M2**, it undergoes the following stages.

a) Suspension of process on **M1**.

b) Check pointing it, *i.e.* collecting snapshot of process on **M1**.

c) Transfer of snapshot to **M2**.

d) Initializing system data structures on **M2** and reestablishing communication with outside world.

e) Restarting it on **M2**.

Usually, Phase **b** and phase **c** are performed in parallel to eliminate the need of buffering. However, in case for fault tolerance systems [Sri91], these can not be

done in parallel because here checkpoint file is prepared periodically even if transfer is not needed.

3.2 Inherent problems with migration

Process migration has some inherent bottlenecks. There is a time lag between suspension and restarting, during which the process is not active. Thus it can not respond to outside events and therefore demands the queuing of essential events.

Moreover process faces some unexpected events due to its reincarnation. For example, when a file is successfully opened by a process, it exists throughout its execution. But file cannot be seen after migration, if another process deletes it during its migration.

Another problem is it may not access resources in the same way it does on the original machine. This is due to subsystems, which are not distributed in nature. Pipes, semaphores etc. are designed by keeping uniprocessor system in mind. They are not suited to distributed system without major modifications.

Another problem is fragality. All parts of the OS have to be considered while introducing migration facilities. Even after introduction, every subsystem has to consider its effect on migration, whenever it needs modifications.

3.3 Migration vs. remote execution

There are some other approaches that can be used in place of migration. Rollback and shadow paging are alternatives used in fault tolerant systems. We will not be discussing them, as they are out of scope for this work.

Remote execution is an alternative approach for load balancing scheme. Butler [Nich87] and REM [Shoja87] are examples systems employing remote execution. With experience from Butler [Nich87], a load balancing system based on remote execution, Nicols noted that addition of migration facility will make his system convenient.

Migration has certain advantages over remote execution. Remote execution programs run entirely on remote machine where as migration may occur at any time during the execution. Migrated programs run partially on a machine till the migration and may even come back to the original machine after few migrations. Generally, programs are invoked explicitly for remote execution, whereas users remain unaware of process migration. Butler is however an exception to this.

Computation intensive jobs, *i.e.* jobs requiring low interaction are large in number and incur no additional overhead due to migration.

3.4 Migration mechanism

Migration mechanism is divided into strategy and policy.

1) **Migration Policy:** It concerns design decisions like when to migrate and where to migrate. It consists the way of selecting idle machine, time of migration and eviction.

2) **Migration strategy:** It consists of implementation of check pointing, transfer, setup and maintenance.

Such a layering offers certain advantages, like code modularity. For example, same bottom layer can be used for different policies of upper layer like load balancing and resource availability.

3.5 Migration policy

Policy consists of following decisions.

1) **Transfer policy:** A decision is to be made, whether some process needs migration or not. It answers the question, *When to migrate?*.

2) **Location policy:** Once the transfer policy has decided to get rid of a process, the location policy has to figure out *Where to migrate?*. It involves selecting a suitable host for migration based on the load and other factors.

3) **Selection policy:** More specifically, this is job selection policy. It answers the question, *What to migrate?*. and selects the task to migrate.

3.5.1 Policy characteristics

1) **Localized vs globalized:** Policy can consider information only at local site or can depend on information at all sites in processing pool.

2) **Centralized vs distributed vs hierarchical decision making:** If decisions over network are taken at a single machine,then it is central approach.If decision making authority is distributed on all sites, then it is distributed approach.One compromise is dividing the network into groups and subgroups and giving authority over them to one site per group or subgroup. Subgroup authorities subdue to group authority. This is called hierarchical decision making.

3) **Static vs adaptive** If policy changes according to current state, then it is adaptive or dynamic policy, otherwise it is a static policy. e.g. A dynamic policy based on the system's load will implement the following strategy.

   If system's load is more than threshold for the recent intervals, it increments threshold.

4) **Receiver oriented vs sender oriented approach:** These approaches differ in taking initiative for migration. In one method, an idle or one, that has plenty of resources announces its availability, and informs it has little to do and is ready for extra work. In another approach, a machine overloaded or that hasn't resources requests another machine to take its work.

5) **Heuristic vs deterministic:** Policy has some intent like load sharing to migrate a process. If decisions made by a policy will certainly lead to improvement to the intent, then it is called deterministic policy. These kind of decisions are very difficult unless the future demands for resources are known a priori. Since future demands of processes are not known a priori, heuristics based approach is used.

3.6 Migration strategy

Migration strategy needs to consider the following issues.

1) **Residual dependencies:** Complete state is big and if migrated in totality, will consume large system resources. Thus only part of state is migrated which is absolutely necessary for running the process on other machine. These type of state slicing leave residual dependencies. A good migration strategy should reduce the volume of residual dependencies, as they have to be communicated between machines.

2) **Memory:** Memory image of process can be transfered either by freezing or on demand.

   Freezing: By transferring the entire image at once, residual dependencies are minimized at the cost of large overloads in initial setup.

   On demand: In this approach, image is copied from one machine to another, only when there is need. This method is similar to copy on write on Mach or lazy swapping. Many times, it saves the cost of transferring unused segments of memory, but comes with additional overhead of determining whether memory needs to be copied or not at runtime.

3) **Files:** The migrated process needs to access the same files in the similar way as it was doing before migration. There are many approaches to maintain the state of files after migration.

   Transferring entire state to destination: In this approach, before restarting process, entire file information has to be received and reestablished. [PeLi95] followed this approach.

   Dummy (shadow) processes: Here a process is created on the original machine that does file operations on behalf of the migrated process. Migrated process redirects its file operations to dummy one and receive results from there. Condor [LiMa92] followed this approach.

State-full file servers: For achieving higher transparency, state is maintained at file server, unlike SUN NFS. During migration, server will be informed of the state of file handles. Using this type of approach, file pointer sharing across machines can be achieved. The sprite operating system [DoOu89] achieved this with its own file system.

4) **Shared memory:** Distributed shared memory has to be implemented, in a manner independent of the machine. Munin's system and Iva are two such successful systems.

5) **Mutual exclusion:** There are many algorithms developed for distributed synchronization and mutual exclusion [Meak87]. e.g. Mechkawa, Rangwaala, central server and ring based algorithms .

6) **Communication:** Message handling is complicated task, because while migration some messages may be half way through transfer.There are three possible approaches to follow to handle messages [ChLu89] .

Message Redirection: If process p on **M1** has a channel c to communicate with some other process on machine **M2** before migration and it migrates to machine **M3**, then it opens channel d from machine **M3** to **M1**, which is associated with channel c between machines **M1** and **M2**. **M2** is unaware of p's migration, it uses channel c as previously. Machine **M1** redirects it to channel d for **M3**. The scheme is transparent for **M2**, but each communication passes through an extra loop.

Message loss recovery: After migration, **M2** is informed about p's migration which then establishes a new channel c' between **M2** and **M3**. All the messages in transit during migration are recovered from c on **M2** and sent on c'. The channel c is then closed.

Message loss prevention: In this approach, **M2** is informed of migration, before migration of process p. **M2** then queues messages on channel c till the migration is completed. After migration **M2** is again informed by **M1**. Channel c' is then established in place of c.

7) **Authentication:** As many resources are shared among machines, authentication is required for each new connection and authorization has to be verified for every request. Kerboros [SNS88] is an example of authentication services.

8) **Process relationships:** It is desirable to have the the same parent, and children after migration. This can be achieved by modifying the process table entries to give a unified view.

9) **Signals:** There is no way of sending signals to processes on other machines except by messages. So signal redirection has to be done to processes' current location.

10) **Naming:** Resources have to be named uniquely on all machines and should be identified as same resources before and after migration.

Chapter 4

Design issues

Before discussing the design details, factors influencing our design are given.

4.1 Influencing factors

4.1.1 Working environment

We implanted our system using SUN-3 workstations in IITK computer science department connected by ethernet in a local area network. Three of them have disks and others are diskless workstations.

Some features of our environment:

- All workstations are running autonomously.
- They are based on Motorola 68020 32-bit microprocessors providing 32 bit address space.
- They run SunOS, which is enhanced version of 4. 2 BSD and 4. 3 BSD [Leff89] Unix systems, with some features from AT&T's system V. 3 UNIX [Bach91].
- No source code is available for SunOS kernel, but it is configurable. The `makefile` to generate `vmunix` is available to provide support for driver development, thus the kernel can be modified.

- The `init_sysent.c` file is available, thus we can add system calls. Wide networking support is available.
- There are many utilities like `adb`, `kadb` and `nl`, that are useful to work with kernel executable file.

4.2 SunOS environment

SunOS is an enhanced version of BSD Unix. It differs with standard Unix in some aspects. We describe here the Sun process address space, kernel data structures and other related information.

4.2.1 SunOS process address space

SunOS process address space is divided into logical segments called regions. Each region is a contiguous area of virtual address space with in the process image. segments can be shared or protected across processes. Each process has at least three regions, namely text, data and stack. Text segment contains the machine instructions that form the executable code. This segment is read only, neither grows nor shrinks. Data segments contains the storage for programming variables, strings, arrays and other data. It has two parts, initialized data and uninitialized bss. Data segments are modifiable. Third segment is is stack segment, that grows or shrinks as subroutines are called.

SunOS supports shared library concept. Two processes using same library code can map this segments at run time using dynamic linkage. One possible address space layout for a SunOS process is shown in Figure 1.

4.2.2 Implementation of processes

Processes are active entities in SunOS. Process state is shown in Figure 2. Every process has a user part and a kernel part. When system calls are invoked, kernel part of the invoking process becomes active and gets executed. The kernel maintains two key data structures related to processes, process table and user structure.

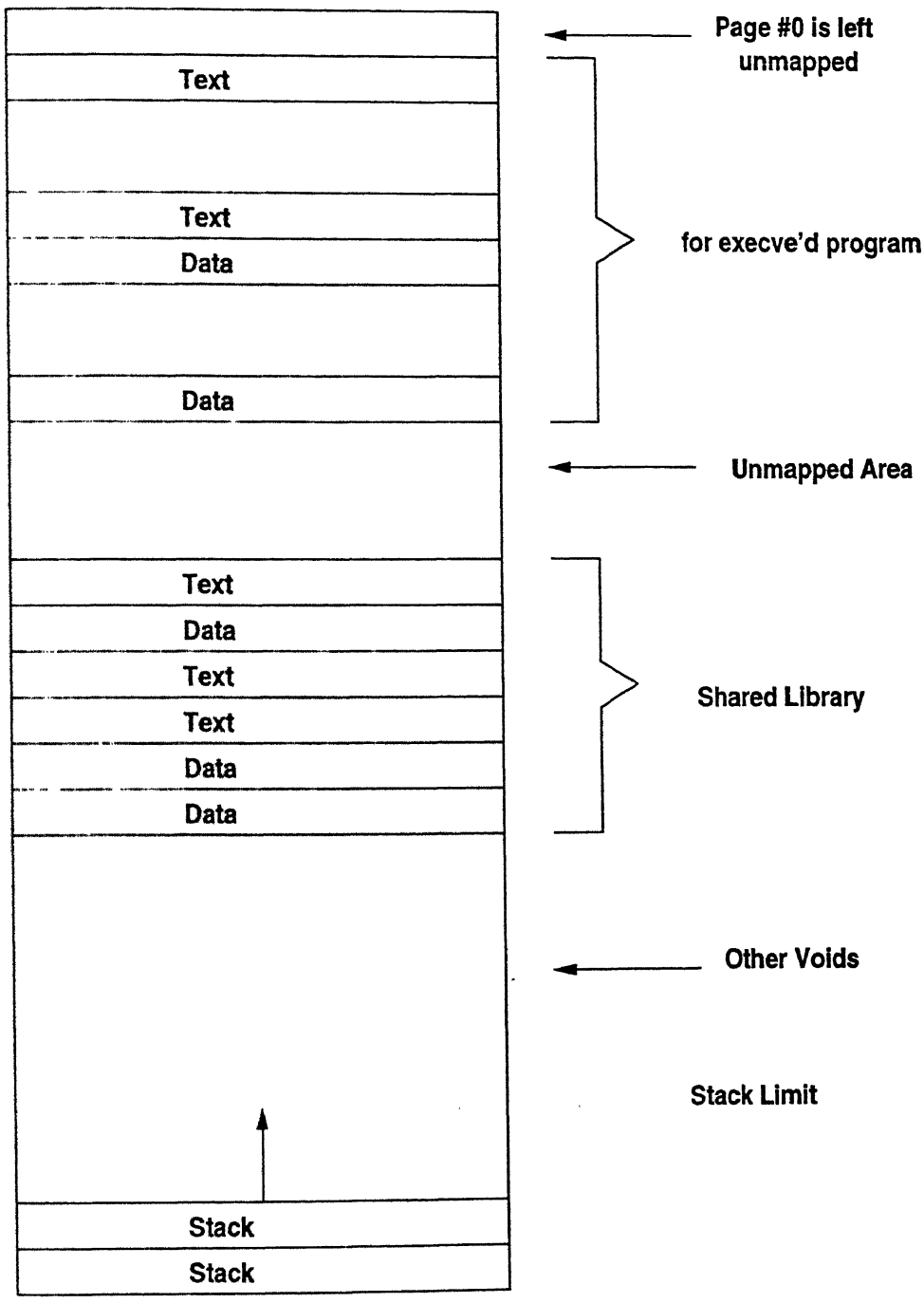


Figure 1: SunOS process address space layout

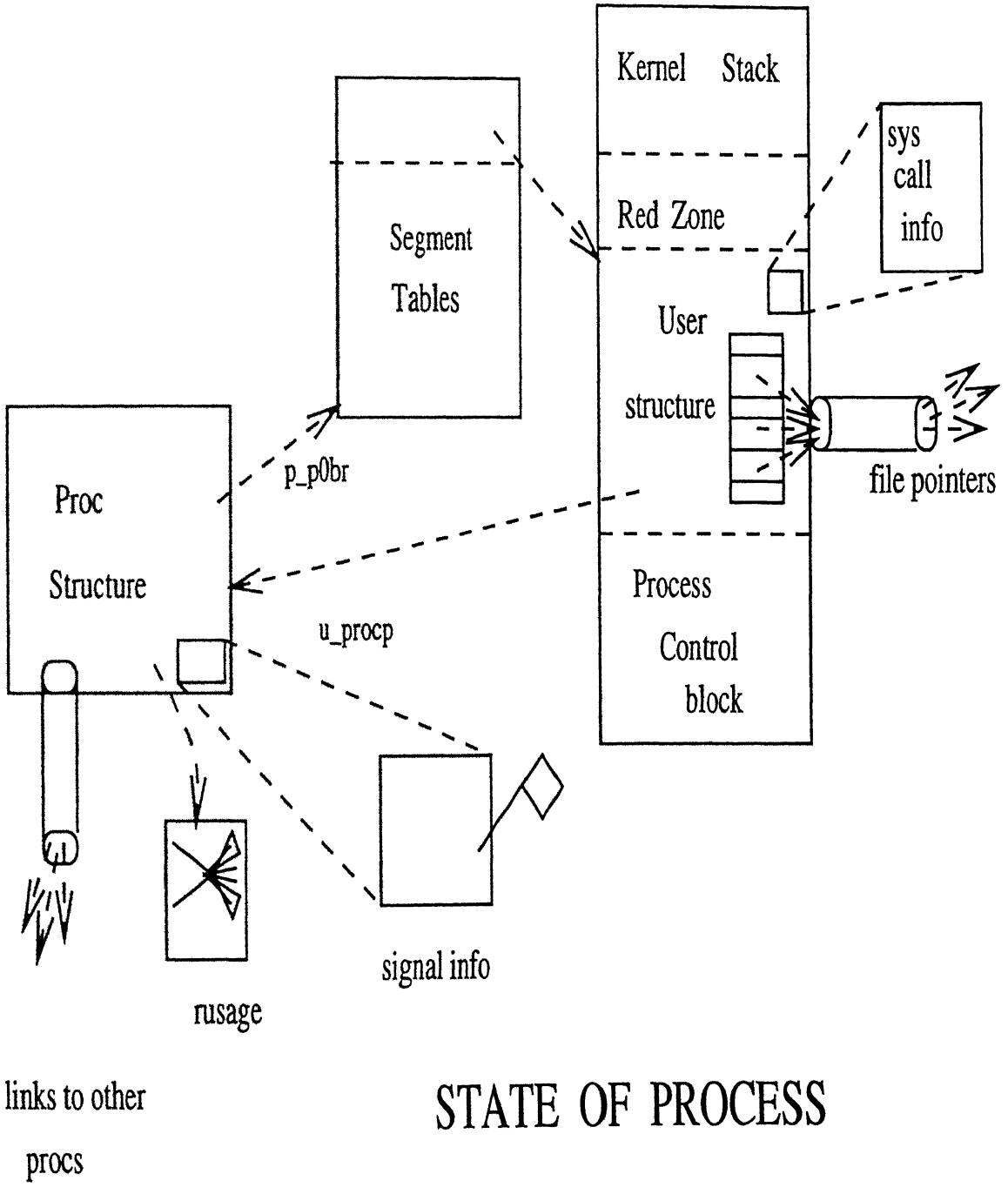


Figure 2: State diagram of process

Each SunOS process has an entry in the kernel process table that points to a process structure. Process structure contains process state that must reside in the memory even though process is swapped out. Various fields are as follows:

- Identification ids: Various ids like `pid`, `uid`, `ppid`, `euid`, `gid`, `pgid`.
- Scheduling parameters: Process priority, amount of cpu time consumed, nice value etc.
- Memory image: Pointers to process region table entries of text, data and stack.
- Signals: Masks showing signals being ignored, being caught, being blocked etc.
- Miscellaneous: Events being waited for, alarm details, pointers to rusage structures, pointers to other related processes in process table.

User structure, also called as *u-area*, contains process information that is not needed when process is not physically in memory and runnable. It includes the following information:

- File descriptor table: Descriptors for file related system calls.
- System call state: Arguments and return values of current system calls.
- Process control block: Hardware specific.
- Kernel stack.
- Register context.
- Pointers to process structure and to different vnodes.
- Timing and statistics related information.

There are some other important structures to be maintained for each process.

- User credentials, ucred.
- User resource usage structure, rusage.
- Per process page tables.
- File table information.

CENTRAL LIBRARY
KANPUR
Acc. No. A 123257

4.2.3 Process's life line

Various stages during lifetime of a process are as shown in Figure 3.

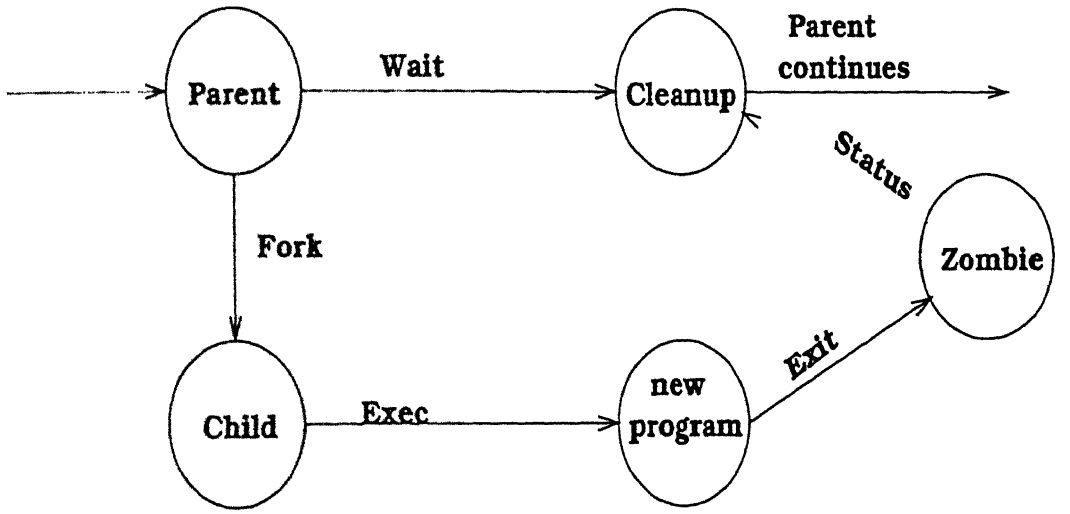


Figure 3: State transition of a process during its life

Every process other than process 0 is created by a call to `fork` system call by parent process. At birth time, child process inherits all state from parent. To overlay its state by a new state in order to perform some other task, it calls *exec* system call. At the end of its execution or in the case of abnormal conditions, it calls `exit` system call. After exit, child's occupied resources are released and there is no more execution in it. But its process table slot will be kept for it, until its parent calls `wait` system call. This state is called *zombie*. After `wait` system call, child's status is given to parent and its process table slot is also released. If parent dies without waiting for its children, children become *orphans*. Process 1, `init`, becomes the parent process of them and waits for them.

4.2.4 File maintenance

On each machine, state of all open files is maintained in a global table. Every opened file by a process is associated with a descriptor. System calls like `create` prepare

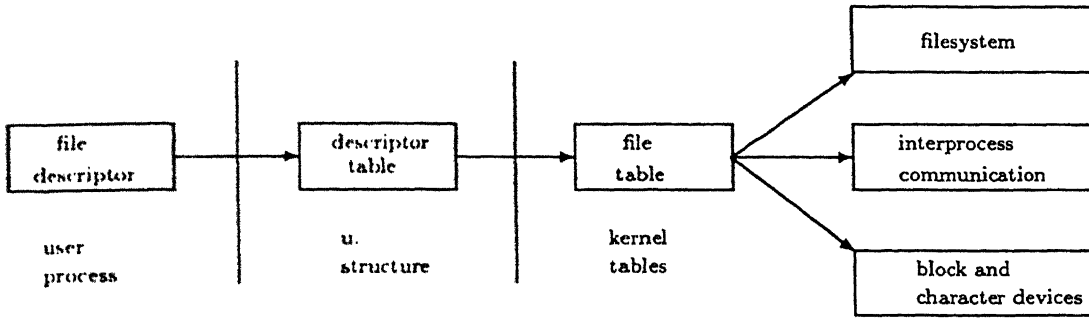


Figure 4: Referring a file from file descriptor

an entry into global file table and obtain a pointer and store it in file descriptor table, a part of u-area. System calls that change state of opened file uses that file descriptor. Reference to a file from a file descriptor is shown in Figure 4.

File is an abstract entity in Unix. File structure consists of a generic field *fdata* and some generic operations. If the file type is *vnode*, then *fdata* is a pointer to *vnode* structure. If it is a *socket*, then *fdata* is a pointer to socket structure.

In the *vnode* structure, there is again type field, that identifies devices and regular files. In the *socket* structure, again there are generic fields, protocol control block and protocol switch structure. These abstractions will continue up to device driver layers.

4.3 Design details

4.3.1 Naming

A uniform naming scheme is needed for resources to be treated same on all machines. Name has to explain its details like home machine. Processes are to be identified both locally and across all systems. One immediate answer is identifying them with `(machine, pid)` simply `mpid`. But there are some problems with this scheme. When process with `pid` *p1* on machine **M1** migrates to machine **M2** and gets identified as process with `pid` *p2*, then this information need to get propagated to all other processes related to it. Otherwise these processes may potentially communicate

with a newly created process with pid *p1* on machine **M1** assuming it to be the old process itself. Allocating a slot to new process and keeping it in a special state like zombie is a solution resulting in wasting those slots.

The reason for this situation is the same *mpid* is used for more than one process in different time spots of system. The pid may be reallocated but time passed away will not recur. Hence for any unique naming scheme, time is the best choice. A same principle is used for vnode generation. Vnode name is combination of *inode*, generation time and machine name. Since at one time instant, only one process may be created at a machine. So only (machine, generation time) fields are sufficient. We call this as *uq_pid*. But a mechanism to map Unix pids from these baked pids is needed. One method is maintaining mapping store on home machine for each process. But this store has to be changed after each migration. Another method is maintaining a global process table, either at central server or by distributing the store and management and it has to be updated properly, whenever there are changes.

4.3.2 Communication paradigm

There are three paradigms for designing distributed systems.

- **Client server model:** Servers manage resources and clients are resource users. Servers wait for request by clients in passive mode. When client contacts, connection is established and communication protocol proceeds.

- **Object oriented model:** Resources are treated as objects and resource usage is implemented as methods on object. Underlying communication is hidden in object implementation and user gets the feeling of accessing local object. Mach follows similar approach. Since each resource is treated in a similar manner and identified with a port, resource access is done by sending messages to its port.

- **Agent oriented computing:** In previous models, processing *i.e.* computation will be done locally on the data that came through messages from the

site resource is located. In this model, computation will be carried in messages and done at resource's location. The programs travel in the network based on the need of resources. Traveling agents is one example. This is different to RPC [BiNe84], since in RPC we send only arguments to computation and get results back, but the code does not travel across network.

We followed the first method, since it is directly available either with sockets or RPC. We are selecting stream sockets for our application, because extra framing, reliability checking and duplicate detection etc. are not necessary.

4.3.3 Software structure

We have divided our software into two layers: migration policy and migration strategy.

Kernel level vs User level: A modified kernel is more transparent than any user level software in accessing system resources. With modified kernel, old executables will work. Moreover, we can access complete state of process using `proc` and `user` structures. While a user level software can not access complete state of system. *e.g.* socket buffers, the major disadvantage of kernel modification is that it is not portable without changes. A new technique in user level software is using debugger interface of `ptrace()`. This technique is used in [PeLi95]. But it incurs high overhead due to additional context switches. Source code that can be used as starting point for our work is available from previous M. Tech thesis [Parik92]. Part of the code that is useful for us is at kernel level. Due to this reason and from above advantages, we decided to provide migration at kernel level. We, however, implemented load balancing policy at user level.

Under part of migration strategy, there are some system calls like *pm_fork()*, *pm_migrate()*, *pm_receive()*, *pm_exit()*, *pm_wait()*, *pm_getpid()* and *pm_getppid()*. All these system calls interact with process migration servers that are located one per machine. Actual implementation of these is explained in next chapter. Server code is responsible for accessing the resources at other site.

4.3.4 Load balancing

Our policy is to balance load on all machines listed in our processing pool. There are many measures of load. Some are given below.

- Cpu utilization
- Average run queue length
- Memory available
- Disk operations
- System call rate
- Context switch rate

Average queue length is the best estimate of load. If it is high, there is high probability that all other parameters are high. Every measure has one threshold to identify the high load on machine. Methods to identify load on a machine and useful guidelines are given in [SPT]

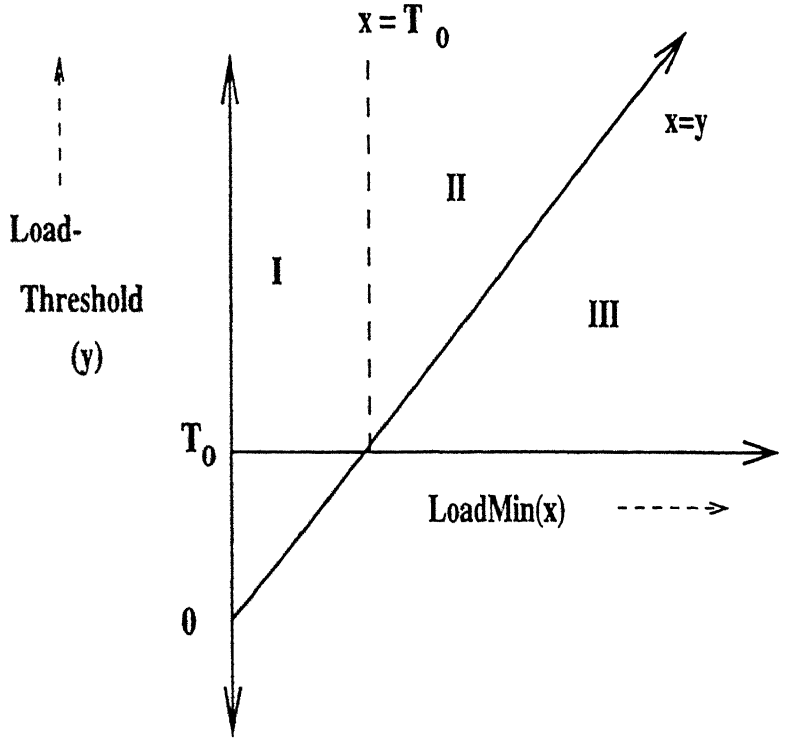
4.3.4.1 Details of policy

Policy decisions can be catagorised as follows.

- **Transfer policy:** If the current load on a machine is more than threshold value, then migration will be tried to other machines.
- **Location policy:** Idle host will be selected, based on the same criteria, *i.e.* if load on remote host is less than threshold, then that host will be selected. Hosts will be tried in round robin order, in order not to swamp a host too much.
- **Selection policy:** Jobs, which have to be migrated are one of the two types, namely,

  1 Local jobs which have not yet started execution.

  2 Foreign jobs earlier migrated to this machine. This will be the case when load becomes more than the threshold.



T_0-intial threshold value

I - Migration allowed.

II- Migration allowed and parameters

will be changed

III - Parameters will be changed

Figure 5: Semantic view of policy

4.3.4.2 Policy characteristics

Our transfer policy is perfectly local, it only depends upon the load on current host. Selection policy is also local. But location policy is not local, and can not be. Decision making is at all sites. Each site makes decisions for the jobs, running there. Our policy is sender initiated one, *i.e.* sender requests receiver to lessen its load.

Our policy is not deterministic, it only tries to lessen the load burden on some machines, but it is not strict balancing for all sites. Hence, load sharing is better suited word for our policy than load balancing. Policy is dynamic. Decision parameters are changing based on loads on hosts. Threshold is changing based on load at all sites.

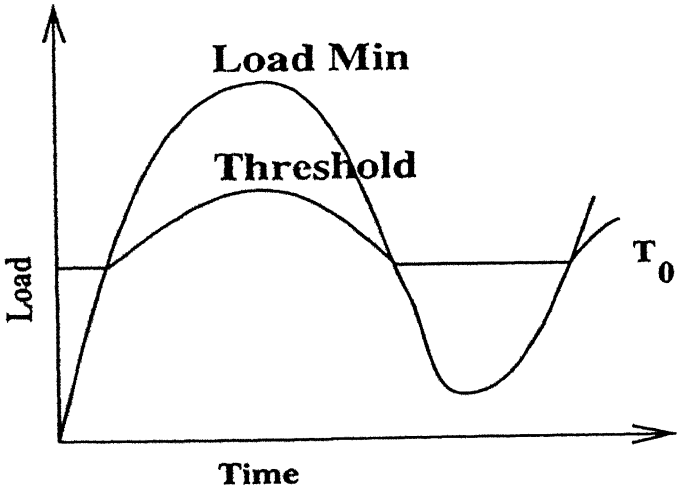


Figure 6: Loadmin vs. loadthreshold

4.3.4.3 Policy parameters and their interrelation ship

- **Load threshold:** This is the crucial parameter for decision making and set to a well analyzed value initially. If load at a site is less than this, decisions take place. It changes its value based on LoadMin.
- **LoadMin:** Minimum of current loads at all sites in pool.

- **SelfLoad:** Load on current host.
- **Slowdown factor(α):** Changing load threshold hastily can lead to some problems like oscillating it or putting too much burden on migration and eviction. It should reduce meandering as possible.
- **Initial threshold(T_0):** It is not a variable, but a pure pre-estimated constant. It is used in decision making as given below.

Load balancing decisions:

If(SelfLoad is less than Load threshold), no migration at all.
If(Selfload > LoadThreshold and LoadMin <= Loadthreshold),
    Do migration to the node corresponding to LoadMin.
If(SelfLoad > LoadThreshold and LoadMin > LoadThreshold),
    No migration will take place.
If(LoadMin <= LoadThreshold and LoadThreshold > T_0),
    LoadThreshold $-=$ (LoadMin $-$ LoadThreshold)$*\ \alpha$
If(LoadMin > LoadThreshold),
    LoadThreshold $+=$ (LoadThreshold $-$ LoadMin)$*\ \alpha$

Schematic view of policy is shown in Figure 5. In regions *I* and *II*, migration is helpful. In regions *II* and *III*, parameters are changed based on current parameters.

The effect of above decisions is shown in Figure 6 for one possible load distribution curve.

4.3.5 Mechanics of migration

Migration is performed by system calls with the help of migration server, *migd*. When a user program calls migrate system call invocation routine, C library puts syscall number in stack and issues a trap to operating system. The system call handler does usual validation, checking of arguments, establishes a connection to migration server that is already waiting in passive mode and then sends request packets to the migration server.

In response to these requests, server forks a child and sets up environment to do migration, if necessary and then calls a system call, that take the process image from remote machine. This code first performs the authentication and informs migration acceptance to the remote machine.

Then entire execution state is then transfered from client to server. Re establishment of its execution state is done there and starts execution there. Server *migd* forks and calls the *pm_receive* on receiving *pm_request*.

Migrate system call is used for migrating a task to other machine. System call *pmrcv* is used be the *migd* server to respond to the migrate system call. *Getpmpage* and *putpmpage* have similar association. *Pmopen* and *pmcreate* are used to open a file instead of old calls open and creat. *Close* and *dup* are modified to get the same effect of dup and close after migration. *Pmexit* and *pmwait* are here for maintaining old semantics of wait and exit for migrated processes. Pmpage transfer routines are used for maintaining global process table.

4.3.6 File maintenance

Dummy process approach is most costly, as it involves more communication between machines. File servers available in our lab use SUN NFS and do not maintain state of open files. In our approach, therefore, we use the following methodology. At the migration time, file states are transferred to the peer. An NFS file is identified with same name on all machines. Hence, we need a mapping from file descriptor to file name. Each file descriptor maintains an offset into a file. After each read and write, file offset is changed accordingly. File sharing by two descriptors is done by maintaining a common offset for them. This is maintained in global file table. File sharing between processes is called inter process file sharing. This is possible by fork system call. Inter process file sharing is not manageable without maintaining state in the global file tables. Stateful file server is a solution for this problem. Intra process sharing is between descriptors of same process. This is possible by dup system call. Intra process sharing can be done by maintaining auxiliary information for each process.

4.3.7 Memory transfer

This is not developed in this thesis, but has been taken from Parikh's implementation [Parik92], since it is already implemented and meets our purpose.

This was originally developed for rfork model [Parik92] to execute the programs in parallel using a network. One implication from this model is checkpointing the address space of the same process and hence migration has to be initiated by the process itself.

4.3.8 Process table maintenance

Process tables are needed at all sites. Central server method has critical failure problem. Distributed shared memory method is selected for maintaining these tables.

Memory occupied by process tables are divided into logical pages.

A page is owned by a single host and physically exists in its memory. All other hosts know the ownership relation for all pages. Thus an explicit synchronization mechanism is not needed, since a page ownership can serve as a token to access a page.

Initial page allocation:

- Method 1: Each machine will be given ownership for the pages for which $p \,\%N = m$, where p is the page no, N is the total number of machines in the system and m is the machine address between 0 and $N - 1$.
- Method 2: A single machine will be given ownership of all pages.

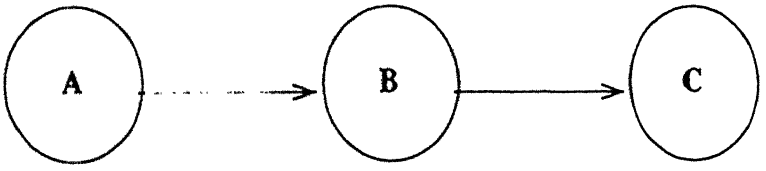
Owner links: A machine may not have ownership for some pages, but it remembers the owner of every page. But this may not be correct and is only a hint to find exact owner and so this field is called probable owner(probowner). This hint field is updated in following cases.

- When M1 transfers page p to M2, M1 updates as probowner(p) = M2.

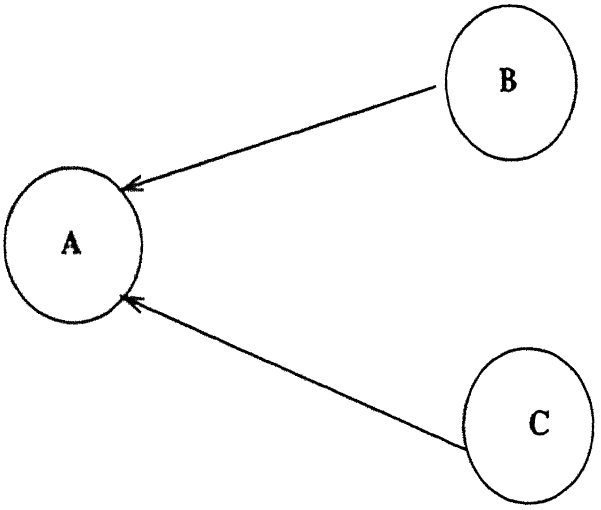
- When M2 requests M1 for page p and M1 has no ownership on page, M1 informs its probowner(p) to M2, and also passes the request to its own probowner(p).
- When M1 requests M2 for page p and M2 informs M1 that its owner is M3, then M1 updates probowner(p) = M3.

e.g., initial links and links after A gets page are shown in Figure 7.

An Optimization: Initially no kernel memory will be allocated for pages. Owners allocate memory only after request, but they maintain a flag for them.



Before A requests page



After A gets page

Figure 7: Owner links

Chapter 5

Implementation

5.1 Data structure for process migration

The kernel needs to maintain information about migrated process to behave like non migrated process after migration. This information need to be made available either in Proc structure or in U-area. Since SUN OS source is not available , extra fields can not be appended. However the U-area has a field u. u_xxx[2], that is not used for any purpose. Hence, state required for migration can be kept using this field. A structure named pm_struct_per_process is maintained for each process and pointer to this structure is stored in U-area fields.

```
struct pm_struct_per_process{
        int       pm_home; /* inet address of the home machine */
        u_long    pm_source; /* inet address of the source machine */
        u_long    pm_host; /* inet address of the host machine */
        u_long    pm_uqpid; /* unique process_id */
        char      *pm_fmap[NOFILE]; /* descriptors to name map */
        short     pm_dmap[NOFILE]; /* file table entry for process */
        int       pm_stat; /* status of process */
}
```

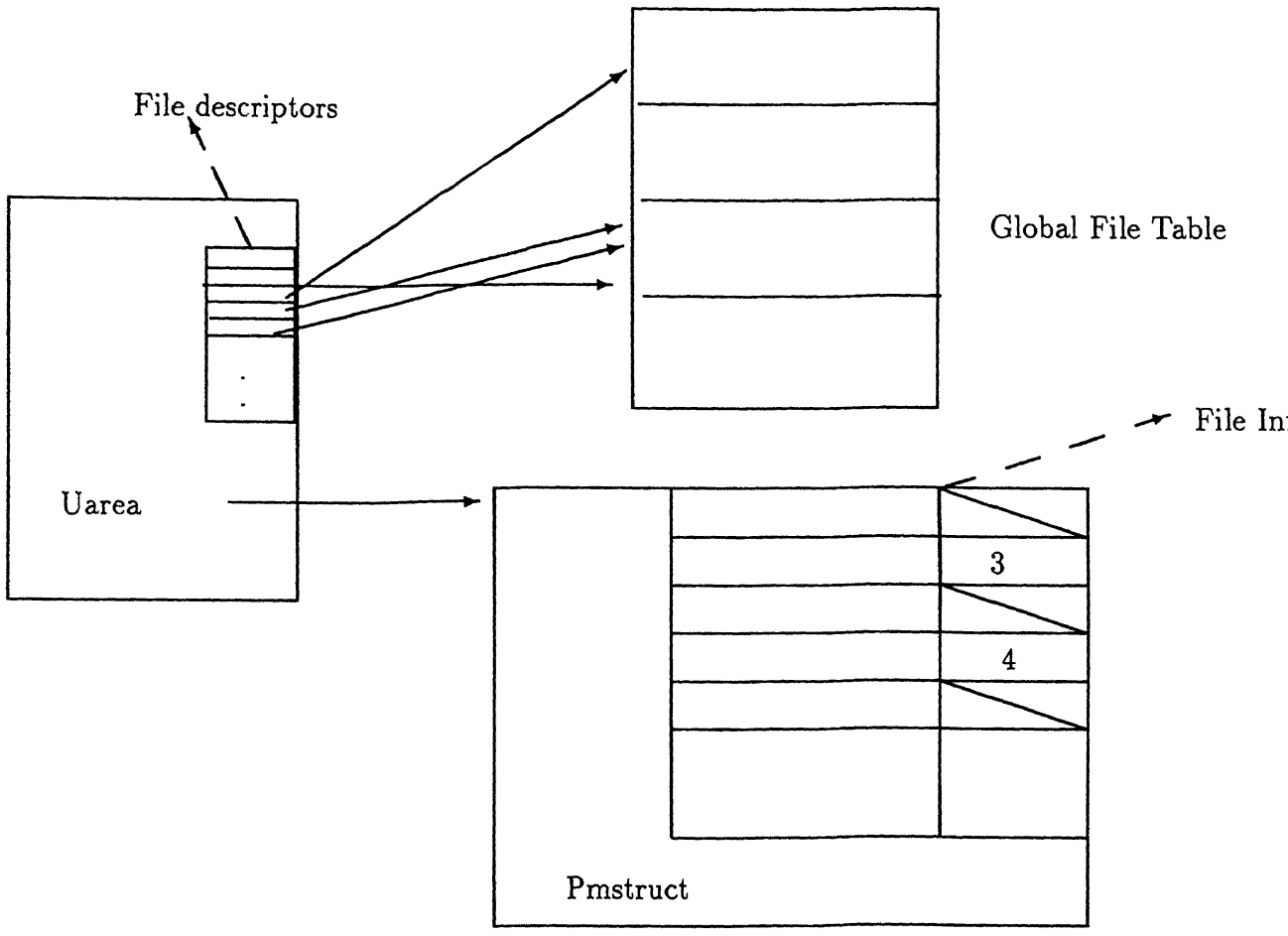


Figure 8: Dup chains

5.2 File handling

In our approach, we store the name of the file after it is opened in Pm_fmap and perform intra-process sharing of files by remembering extra links.

After each dup system call, Pm_dmap chain is modified.

For example, when dup(1, 3), dup(3, 4) are executed, the dup chains get modified which are shown in Figure 8.

Approach adopted for establishing state of files after migration is given below.

Client's role: Client sends some file related information, while it is passing pm-struct to the server as given in the structure.

In addition, for each opened file it also sends,

- file descriptor
- file offset and type from global file table.

The server recreates file using the received information as follows.

- For each file descriptor received, receive all file related information(file offset and type).
- For each file per dup chain, open file in required mode, if type is a regular file
- Seek to required offset
- Dup it to fd, where it was before migration and close current fd.
- Follow the dup chain and dup to all file descriptors in the chain.

5.3 Process table maintenance

Processes that are not using migration facilities will continue to work and their maintenance is through local process table as earlier. But processes that do migrate are maintained by using another global process table, in addition to the table at each site. The global table is maintained by shared memory. This table occupies one or more pages. A process accesses them by using get_pm_page and put_pm_page routines. Circular waiting of machines for these pages is avoided by accessing them in order. Get_pm_page and put_pm_page routines handle page ownerships and transfers.

5.3.1 Control abstraction of *get_pm_page*

input : pmpage /* page no to be accessed */

1. While(our machine is not owner for pmpage) {

   Open a socket in tcp/ip mode.

   Connect to current prob owner site.

   Send the page no.

   If reply is positive, get the page and break.

   Else update prob owner from reply.

   }

2. If our machine is the owner and memory was not allocated, allocate memory to it.

3. Return the page.

5.3.2 Control abstraction of *put_pm_page*

1. Read the requested page no on connected socket.

2. If owner, send the page.

3. Else send the prob owner name.

4. Change probable owner to the client's machine.

Each migrated process corresponds to a slot in global process table. Data structure called as pm_proc is similar to proc and contains fields pointed to the structures of related processes. In Unix, all parent processes are also of same type, but here parent process is either a migrated one or conventional Unix process. Hence, parent pointer is either into pm_procs or to Unix_parent structures that are also maintained in shared pages across machines.

Suppose a parent dies without waiting for a child, then child becomes orphan and it is attached to the process 1, i.e., init process. But a separate approach is followed here in handling orphans, because of difficulty in injecting code into init. All routines that insert, search or modify routines are similar to corresponding routines of proc structure maintenance. But delete routine is different from corresponding one. This is needed as part of wait system call.

5.3.3 Control abstraction of deleting slot

input: slot p;

- 1. Put all of it's children's state to orphan.
- 2. For all slots corresponding to zombie children, call this routine recursively.
- 3. Modify the related links in other slots.
- 4. Return the exit status.

5.3.4 Control abstraction of *exit* routine

input: slot status;

- 1. If the process in that slot is orphan, delete the slot.
- 2. Otherwise, put it in zombie state.

In order to purge the slots of orphan, exit handler will call delete routine, if that process is already an orphan.

5.4 Load balancing

Rstatd, a daemon is running on top of RPC one per machine. It provides all statistics related information of kernel. Whenever a decision has to be made, RPC client handles have to be created and contacted with rstatds at all machines given in a pool file that contains list of all machines allowed for migrating. On the information collected from all of these machines, algorithm that implements policy is applied as explained in previous chapter.

5.5 Migration procedure

Client calls migrate system call and server communicates with it using `pmrcv` system call.

Control abstraction of migrate:

1. Open a tcp/ip socket to server.
2. send request and user credentials. Prove authentication. If reply is not positive return error.
3. Send the kernel data structures proc, u, ucred, rusage and pmstruct.
4. Send all segments of memory image.
5. Send state of all open files.
6. Modify the fields in process table accordingly.
7. return.

Control abstraction of pmrcv

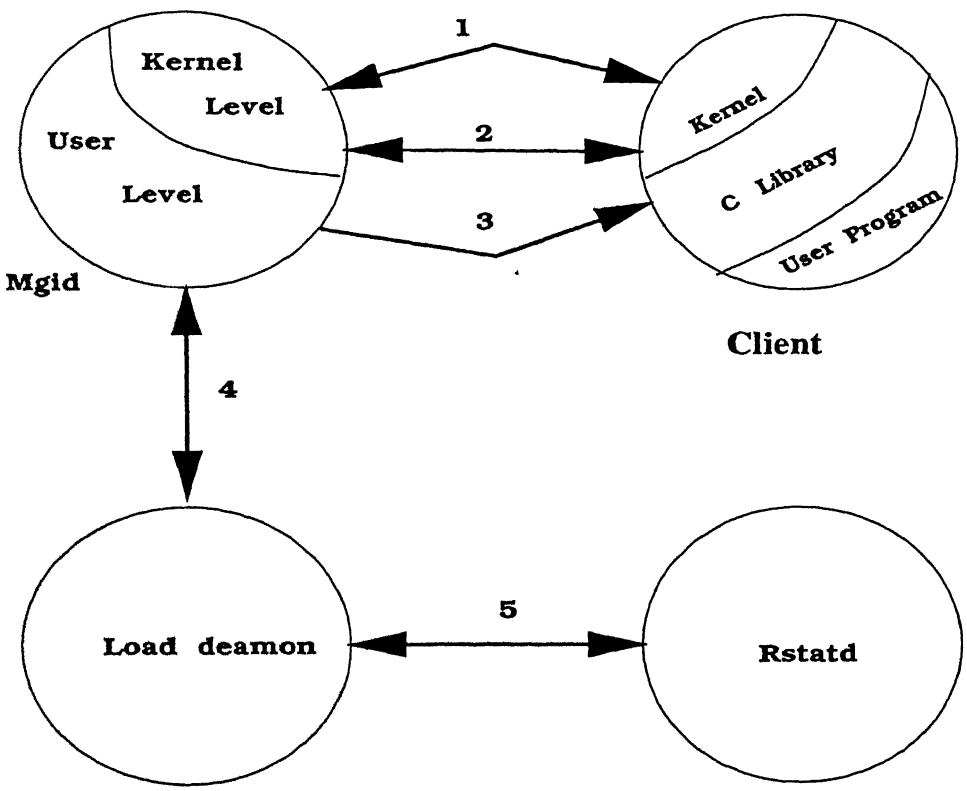
1. read request and user credentials. check authentication.
2. Read the kernel data structures proc, u, ucred, rusage and pmstruct.
3. Receive all segments of memory image.
4. Execve with text, data and stack.
5. Attach other segments.
6. Read Other file related information and reestablish the state.
7. return.

5.6 Authentication

An initial authentication is done to check whether a user is allowed to do migration. Another hack for not to be cheated by any user is creating a dummy file with restricted permissions and making that user as owner of it and asking client to do operations allowed to him.

5.7 Overview of implementation

Client program communicates with server for either on migration protocol or on page transfer protocol. Loadd collects information from rstatd running on other machines and selects a host for migration. This is conveyed to migration server `migd`. Communication paths are shown in Figure 9. In this figure, channel 3 represents the signal flow, channel 5 represents RPC messages while all other represent communication through sockets.



1 **Server and client communicate using migrate() and pmrcv()**

2 **Server and client communicate for page transfer**

3 **Mgid informs client to migrate**

4 **Idle host request and reply**

5 **Load statistics request and reply**

Figure 9: Communication between various components

Chapter 6

Conclusion

6.1 Work done

In this thesis, the design and implementation of a migration strategy and migration policy has been done. In this strategy, the facility to migrate opened files and process relationships are provided. The migration strategy ensures that opened files and process relationships are preserved after migration. File maintenance is provided in state forwarding approach. Process maintenance is provided by maintaining distributed shared momory.

These are implemented with few system calls and a migration server.

The objective of migration policy has been to distribute the load on all systems. The policy is implemented using load daemon. The necessary information for load daemon is provided by migration server.

This system is compatible with `Unix OS`. All computation oriented jobs can be migrated. But complex process, for example, those which use sockets etc won't run in the same fashion after migration.

6.2 Extensions

- Instead of using NFS, stateful file server approach can be used for complete transparency.

- Maintenance of Unix abstractions like signals, pipes, sockets and semaphores has to be added to the system.
- Memory maintenance is from *rfork* model which imposes migration on a process to be done by the same process.
- An alternate approach can be used to do migration by another process.

Bibliography

[Acc86] M. Accetta et al. , *Mach: New kernel Foundation for UNIX development*, Proc. of Summer Usenix Conference, July 1986.

[ArFi89] Y. Arsty and R. Finkel. *Designing a process migration facility*, IEEE computer, sept 1989, pp. 47-56.

[Bach91] Maurice J. Bach, *The Design and Implementation of the UNIX Operating System*, Prentice-Hall of India pvt. Ltd. , 1991.

[BiNe84] Andrew D. Birrell and Bruce Jay Nelson, *Implementing Remote Procedure Calls*, ACM Transaction on Computer Systems, vol. 2, No. 1, Feb 1984, pp. 39-59.

[Bla95] Andrew P. Black et al, *Objects to the resuce or httpd:the next generation opearating system*, Operating Systems Review, Vol 29, ACM, pp. 91-95, Jan 1995.

[Cher88] David R. Cheriton, *The V Distributed System*, Communication of ACM, vol. 31, No. 3, Mar 1988, pp. 314-333.

[Coul94] G. F. Coulouris et al. , *Distributed Systems Concepts and Design*, 2nd ed. Reading, MA:Addision-Wesley, 1994

[Come91] D. E. Comer, *Internetworking with TCP/IP. Vol 1:Principles, protocols and architectures*, 2nd ed. Reading, Englewoodcliffs, NJ:Prentice Hall, 1991.

[ChLu89] Chin Lu, *Process Migration in Distributed Systems*, Technical Report, UIUCDCS-R-89-1488, University of Illinois at Urbana, 1989.

[DoOu89] Fred Douglis and John Ousterhout, *Transparent Process Migration for Personal Workstations,* Technical Report, UCB/CSD 89-540, Computer Science Division, University of California at Berkely, Nov 1989.

[Freed91] D. Freedman, *Experience building a process migration subsystem for unix*, Proceeding of the Usenix winter conference, Dallas, TX, Jan 1991, pp. 349-354.

[LiMa92] M. Litzkow and M. Solomon, *Supporting Checkpointing and Process Migration Outside the UNIX Kernel*, Proc. Usenix Conference, San Francisco, CA, Jan 1992, pp 283-290.

[Gei93] A. Geist et al. *PVM 3 User'sGuide and Reference Manual*, May 1993.

[Leff89] Leffler, McKusick, Karels and Quarterman, *The Design and Implementation of BSD4. 3 UNIX Operating System*, Addision Wesley, 1989.

[Meak87] M. Meakawa, A. E. Oldehoeft and R. R. Oldehoeft, *Operating systems:Advanced concepts,* Menlo park, CA:Benjamin/Cummings, 1987.

[MuSo90] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse and Hans van Staveren, *Amoeba: A Distributed System for the 90's*, IEEE Computer, May 1990.

[Nich87] D. Nichols, *Using Idle Workstations in a Shared Computing Environment*, in proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Nov 1987, pp. 5-12.

[Parik92] Parikh, *Design and Implementation of Distributed Premitives for Workstations*, MTCS-92-23, IITKanpur.

[PeLi95] S. petri H. L. Langendorfer *Loadbalancing and Fault Tolerence in Workstation clusters Migrating Groups of communicating Processes* , Operating systems review , ACM, vol 29, oct 1995, pp. 25-36.

[RFC1094] Inc. Sun Microsystems *NFS: Network File System Protocol specification* Request for Comment, DOC-ID RFC1094, March 1989.

[RYT87] R. Rashid, M. Young, A. Tevanian et al, *The duality of memory and Communication in the implementation of a Multiprocessor Operating system,* Technical Report, CMU-CS-87-155, Aug 1987.

[SiGa94] A. Silberscatz and P. Galvin, *Opearating System Concepts*, Addision-Wesley, 1994.

[Sri91] R. Sriram, *Process Migration for Software Fault Tolerance*, M. Tech. Thesis No. MT-CS-91-14, Indian Institute of Technology, Kanpur, India, Apr 1991.

[Sch95] Harald Schrimpf , *Migration of Processes, Files and Virtual devices in MDX Opearting System*, Operating systems Review, Oct 1995, pp. 70-81.

[Shoja87] G. C. Shoja et al, *REM: a Distributed Facility for Utilizing Idle Processing Power of Workstations*, Proceedings of the IFIP WG 10. 3Working Conference on Distributed Processing, Amsterdam, Oct 1987, pp. 205-218.

[SNS88] J. G. Stenier, B. C. Neuman and J. I. Schiller, *kerberos:An authentication service for Open Network Systems,* Proc. Winter 1988 Usenix conference, San Francisco, CA, Feb 1988, pp. 191-202.

[SPT] *System Performance Tuning*, OReilly Associates Inc. to be checked

[Stev92] W. Richard Stevens, *UNIX Network Programming*, Prentice-Hall of India Pvt. Ltd. , 1992.

[Sun88a] SUN *Writing Device Drivers* Manual, 1988.

[Sun88b] SUN *System Services Overview* Manual, 1988.

[Sun88c] SUN *Network Programming* Manual, 1988.

[Sun88d] SUN *Debugging Tools* Manual, 1988.

[Sun88e] SUN *System Introduction* Manual, 1988.

[Sun88f] SUN *Programming Utilities and Libraries, Release 4. 0* Manual, 1988.

[Smit88] A survey of process migration mechanisms. Operating Systems Review, vol 22, July 1988, pp 28-40.

[Tanen95] A. S. Tanenbaum, *Moderen Operating systems,* Englewood cliffs, NJ:Prentice Hall, 1995.

[Tanen95] A. S. Tanenbaum, *Distributed Opearating systems,* Englewood cliffs, NJ:Prentice Hall, 1995.

[WaPo83] Bruce Walker, Gerald Popek, Robert English, Charles Kline and Greg Theil, *The LOCUS Distributed Operating System*, Proceeding of the Ninth ACM symposium on Operating Systems Principles, 1983, pp. 49-70.

[Zaya87] E. R. Zayas, *Attacking the Process Migration Bottleneck*, Proc. 11th Symp. on Operating Systems Principles, ACM, 1987, pp. 63-76.

Date Slip 123357

This book is to be returned on the date last stamped.

C.S.E-1996-M-RED-PRO